The goal of the assignment is to develop an 8-bit Shift Register with 4 distinct modes of operation –

Mode 0 = Do Nothing

Mode 1 = Shift Right (with MSB = serial right input)

Mode 2 = Shift Left (with LSB = serial left input)

Mode 3 = Parallel Load

This 8-bit Shift Register is to be modeled using C++ classes, taking full advantage of language compartmentalization of different logic units (e.g. D Flip Flop). The difference here is that a focus will be placed on introducing timing, power consumption, along with logic output seen at the wires. A testbench is also created, which will apply test data to the Shift Register and output the contents to an external file, which will detail all of the information and showcase the output.

**Wire Declaration & RT Level Components** (Parts A & B)

The first part of this homework asked for the definition of a wire that can carry a delay (timing), power consumption, and the logic value from a gate output. Once this wire is declared in Part A, it was asked that RT level components (such as gates and DFF’s) be developed in order to create the Shift Register. Two files were created that contain the definitions of this data, and are contained in the *shiftRegister* project folder provided in the .ZIP:

*logicPrimitives.h*

*logicPrimitives.cpp*

The first file, *logicPrimitives.h*, serves to define the construct of the wire model requested, along with the class template of basic RT level components. Some functions are defined inline within the class definitions, while the crux of the logic is defined in the *logicPrimitives.cpp* file. As described in the homework description, all delays are based on the gate input # in question, and the respective delays are set within the constructors themselves.

// Carlos Lazo

// ECE579D

// Homework 04

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

#define MAX2(a,b)((a < b) ? b : a);

#define MAX3(a,b,c) (c>((a>b)?a:b))?c:((a>b)?a:b);

// Define the wire class, which will hold:

// logic value, pwr consumption, and timing

class wire {

char value;

int delay, pwr;

public:

wire () {delay = 0; pwr = 0; value = 'X';}

void put (char v, int d, int p) {value = v; delay = d; pwr = p;}

void get (char& v, int& d, int& p) {v = value; d = delay; p = pwr;}

int delayValue() {return this->delay;}

char getVal() {return this->value;}

};

// Define a 3-input AND gate

class and {

wire i1, i2, i3, o1;

int gateDelay;

public:

and () {gateDelay = 3;} // Gate Delay = 3, since this is a 3 input AND gate.

int delayValue() {return this->gateDelay;}

void AND (wire, wire, wire, wire&);

};

// Define a 4-input OR gate

class or {

wire i1, i2, i3, i4, o1;

int gateDelay;

public:

or () {gateDelay = 4;} // Gate Delay = 4, since this is a 4 input OR gate.

int delayValue() {return this->gateDelay;}

void OR (wire, wire, wire, wire, wire&);

};

// Define the NOT gate

class not {

wire i1, o1;

int gateDelay;

public:

not () {gateDelay = 1;} // Gate Delay = 1, since this is a NOT gate.

int delayValue() {return this->gateDelay;}

void NOT (wire, wire&);

};

// Define the DFF class

class dff {

wire Q;

int clkQDelay;

public:

dff() {clkQDelay = 3;} // Set Clock Delay = 3

int delayValue() {return this->clkQDelay;}

wire get\_Q() {return this->Q;}

void DFF(wire, wire, wire);

};

The corresponding file is the *logicPrimitives.cpp*, which implements all functions described above in the header file. Detailed comments are written within each function to explain the operation, and are all based on the *wire* implementation denoted above. It is assumed that the maximum delay of all gate inputs is the one pushed forward, and for that the power consumption is a unitless measure, being incremented by 1 for each gate that is used to resolve the circuit:

// Carlos Lazo

// ECE579D

// Homework 04

#include "logicPrimitives.h"

// Define the AND function

void and::AND(wire i1, wire i2, wire i3, wire& o1)

{

char av, bv, cv, wv;

int ad, bd, cd, wd;

int ap, bp, cp, wp;

i1.get(av, ad, ap);

i2.get(bv, bd, bp);

i3.get(cv, cd, cp);

if ((av == '0')||(bv == '0')||(cv == '0'))

{

wv = '0'; // Value = 0

// Setup gate delay and pwr based on which input is '0' first

if (av == '0') {

wd = ad + this->gateDelay; wp = ap + 1; }

else if (bv == '0') {

wd = bd + this->gateDelay; wp = bp + 1; }

else {

wd = cd + this->gateDelay; wp = cp + 1; }

}

else if ((av == '1')&&(bv == '1')&&(cv == '1'))

{

wv = '1'; // Value = 1

// When all gates are = 1, calculate 'worst case' pwr and delays

wp = MAX3(ap,bp,cp) + 1;

wd = this->gateDelay + MAX3(ad,bd,cd);

}

else

{

wv = 'X'; // Value = X

// Setup gate delay and pwr based on which input is '0' first

if (av != '1') {

wd = ad + this->gateDelay; wp = ap + 1; }

else if (bv != '1') {

wd = bd + this->gateDelay; wp = bp + 1; }

else {

wd = cd + this->gateDelay; wp = cp + 1; }

}

o1.put(wv, wd, wp); // Place calculated value on output wire

}

// Define the OR function

void or::OR(wire i1, wire i2, wire i3, wire i4, wire& o1)

{

char av, bv, cv, dv, wv;

int ad, bd, cd, dd, wd;

int ap, bp, cp, dp, wp;

i1.get(av, ad, ap);

i2.get(bv, bd, bp);

i3.get(cv, cd, cp);

i4.get(dv, dd, dp);

if ((av == '1')||(bv == '1')||(cv == '1')||(dv == '1'))

{

wv = '1'; // Value = 1

// Setup gate delay and pwr based on which input is '1' first

if (av == '1') {

wd = ad + this->gateDelay; wp = ap + 1; }

else if (bv == '1') {

wd = bd + this->gateDelay; wp = bp + 1; }

else if (cv == '1') {

wd = cd + this->gateDelay; wp = cp + 1; }

else {

wd = dd + this->gateDelay; wp = dp + 1; }

}

else if ((av == '0')&&(bv == '0')&&(cv == '0')&&(dv == '0'))

{

wv = '0'; // Value = 0

// When all gates are = 1, calculate 'worst case' pwr and delays

wp = MAX3(ap,bp,cp); wp = MAX2(wp, dp);

wd = MAX3(ad,bd,cd); wd = this->gateDelay + MAX2(wd, dd);

}

else

{

wv = 'X'; // Value = X

// Setup gate delay and pwr based on which input is '0' first

if (av != '0') {

wd = ad + this->gateDelay; wp = ap + 1; }

else if (bv != '0') {

wd = bd + this->gateDelay; wp = bp + 1; }

else if (cv != '0') {

wd = cd + this->gateDelay; wp = cp + 1; }

else {

wd = dd + this->gateDelay; wp = dp + 1; }

}

o1.put(wv, wd, wp); // Place calculated value on output wire

}

// Define the NOT function

void not::NOT(wire i1, wire& o1)

{

char av, wv;

int ad, wd;

int ap, wp;

i1.get(av,ad,ap);

if (av == '1') wv = '0';

else if (av == '0') wv = '1';

else wv = 'X';

wd = ad + this->gateDelay;

wp = ap + 1;

o1.put(wv,wd,wp);

}

// Define the DFF function

void dff::DFF(wire D, wire C, wire R)

{

char Dv, Cv, Rv, Qv;

int Dd, Cd, Rd, Qd;

int Dp, Cp, Rp, Qp;

D.get(Dv, Dd, Dp);

C.get(Cv, Cd, Cp);

R.get(Rv, Rd, Rp);

if (Rv == '1') {

Qv = '0'; Qd = 0; Qp = 0;

}

else if (Cv == 'P') {

if (Cd >= Dd) Qv = Dv;

else Qv = 'X';

Qd = this->clkQDelay;

Qp = 1;

}

// Catch-all statement, leave Q as it is (e.g. clk = 'N')

else

this->Q.get(Qv, Qd, Qp);

this->Q.put(Qv, Qd, Qp);

}

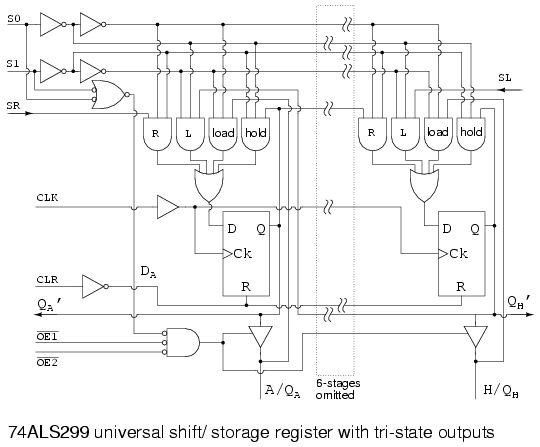
**Shift Register & Testbench** (Parts C & D)

In order to easily encapsulate the data necessary for the Shift Register, two separate files were created to contain all necessary data and logic:

*ShiftRegister.h*

*ShiftRegister.cpp*

The *ShiftRegister.h* file contains all necessary declarations in order to model a Shift Register. Given that the Testbench that will be created takes in data from an external file, the Shift Register has declared characters to read in the data, which is then placed on its own “internal wires” before analysis is performed on the input stream. Values that store the total delay and power consumption of each iteration of the circuit are also noted. The following diagram denotes the way that this Shift Register was created, and shows why the implementation was done as showcased. Please note that none of the Clock Enable signals were implemented in this code:



Here is the code to go along with this diagram:

// Carlos Lazo

// ECE579D

// Homework 04

#include "logicPrimitives.h"

// Define the ShiftRegister Class

class ShiftRegister {

private:

// Declare all private member variables of the Shift Register:

char din [9]; // Parallel input

char pout [9]; // Parallel output

char clk, rst, m1, m0, sri, sli;

wire w\_clk, w\_rst;

int t\_Delay;

int t\_Power;

string in\_str; // Stores input string for printing

string pr\_dff; // Stores previous flip flop values

dff reg [8]; // DFF's that make up Shift Register

wire dTemp [9]; // Define temp DFF wires for computation

public:

// Public functions that will be executed in order:

ShiftRegister(); // Default Constructor

void load\_input (string); // Parse input string into member variables

void update\_reg(); // Based on m1m0 mode, update register accordingly

void set\_output(); // Set all DFF's, based on clock and reset signals.

string output\_reg();// Formatted output generated for the Shift Register.

};

The matching file is *ShiftRegister.cpp*, which defines all of the *ShiftRegister* functions listed above. It is important to note that the update\_reg() function is a Gate Level implementation of the Shift Register. Detailed comments are contained within the code showcased below. Note that the last function, output\_reg(), is used for the Testbench:

// Carlos Lazo

// ECE579D

// Homework 04

#include "ShiftRegister.h"

// Define the ShiftRegister constructor

ShiftRegister::ShiftRegister()

{

for (int i = 0; i < 8; i++)

{ din[i] = 'X'; pout[i] = 'X'; dTemp[i].put('X',0,0); }

// Note that DFF's have Q = 'X' by default, as per that specific constructor

din[8] = '\0'; pout[8] = '\0'; dTemp[8].put('\0',0,0);

clk = 'X'; rst = 'X'; m1 = 'X'; sri = 'X'; sli = 'X';

t\_Delay = 0; t\_Power = 0;

}

// Define the ShiftRegister load\_input function

void ShiftRegister::load\_input(string indata)

{

in\_str = indata;

// The input stream from the text file is as follows:

// m1 m0 | sri | srl | din | rst | clk ==> Information

// 0 1 2 3 4-11 12 13 ==> String Location

m1 = indata[0]; m0 = indata[1];

sri = indata[2]; sli = indata[3];

for (int i = 4; i < 12; i++)

din[i-4] = indata[i]; // din[0] = MSB, din[7] = LSB

// Declare new wire values for rst and clk based on each input

rst = indata[12]; clk = indata[13];

w\_rst.put(rst, 0,0);

w\_clk.put(clk, 50,0); // Assume a clock delay of 50ns.

// Store current flip-flop values:

pr\_dff = "";

for (int i = 0; i < 8; i++)

pr\_dff += pout[i];

t\_Delay = 0;

t\_Power = 0;

}

// GATE LEVEL Implementation of Shift Register

void ShiftRegister::update\_reg()

{

// Declare all necessary wires.

wire w\_sr, w\_sl, w\_m1, w\_m0, w\_din;

w\_sr.put(sri,0,0); w\_sl.put(sli,0,0);

w\_m1.put( m1,0,0); w\_m0.put( m0,0,0);

// Declare NOT wires of input signals m1 and m0.

wire w\_m1N; not n1; n1.NOT(w\_m1, w\_m1N);

wire w\_m0N; not n0; n0.NOT(w\_m0, w\_m0N);

// Declare all necessary gates and wires for mode calculation.

and dn\_gate, rs\_gate, ls\_gate, ld\_gate;

or in\_DFF;

wire w\_dnO, w\_rsO, w\_lsO, w\_ldO, w\_inDFF;

char wv; int wd, wp;

// Perform shift-register operations based on mode inputs.

for (int i = 0; i < 8; i++)

{

w\_din.put(din[i],0,1); // Assume inputs count towards power consumption

// If mode = {0,0}, do nothing.

dn\_gate.AND(reg[i].get\_Q(), w\_m1N, w\_m0N, w\_dnO);

// If mode = {0,1}, shift right.

// If at the 1st shift register, rs\_gate takes in sri.

// If not, then it takes the Q value from the previous shift register.

if (i == 0)

rs\_gate.AND(w\_sr, w\_m1N, w\_m0, w\_rsO);

else

rs\_gate.AND(reg[i-1].get\_Q(), w\_m1N, w\_m0, w\_rsO);

// If mode = {1,0}, shift left.

// If at the last shift register, ls\_gate takes in sli.

// If not, then it takes the Q value from the next shift register.

if (i == 7)

ls\_gate.AND(w\_sl, w\_m1, w\_m0N, w\_lsO);

else

ls\_gate.AND(reg[i+1].get\_Q(), w\_m1, w\_m0N, w\_lsO);

// If mode = {1,1}, parallel load.

ld\_gate.AND(w\_din, w\_m1, w\_m0, w\_ldO);

// Now that all computations are done, the OR of these gates will go into dTemp[i].

in\_DFF.OR(w\_dnO, w\_rsO, w\_lsO, w\_ldO, w\_inDFF);

w\_inDFF .get(wv, wd, wp);

dTemp[i].put(wv, wd, wp);

// Update current operation's delay and power consumption:

t\_Delay += wd;

t\_Power += wp;

}

}

void ShiftRegister::set\_output()

{

// Set all DFF - this is dependant on clock and reset values.

// Utilize values determined in the update\_reg() function.

for (int i = 0; i < 8; i++)

reg[i].DFF(dTemp[i], w\_clk, w\_rst); // Update all DFFs

char tv; int td,tp;

for (int i = 0; i < 8; i++)

{

reg[i].get\_Q().get(tv,td,tp);

pout[i] = tv ; // Set parallel output to DFF Q values

}

}

string ShiftRegister::output\_reg()

{

string pretty\_print;

pretty\_print += "Current ShiftRegister contains the following values: ";

for (int i = 0; i < 8; i++)

pretty\_print += pr\_dff[i];

pretty\_print += "\n";

pretty\_print += "Input = " + in\_str.substr(4,8) + ", Mode = " + in\_str.substr(0,2);

pretty\_print += ", Clk = "; pretty\_print += in\_str[13];

pretty\_print += ", Rst = "; pretty\_print += in\_str[12];

pretty\_print += ", sri = "; pretty\_print += in\_str[2];

pretty\_print += ", sli = "; pretty\_print += in\_str[3];

pretty\_print += "\n\t";

pretty\_print += "After operation, ShiftRegister now contains: ";

for (int i = 0; i < 8; i++)

pretty\_print += pout[i];

pretty\_print += "\n\t";

char bufD[32], bufP[32];

pretty\_print += "Total time delay = ";

pretty\_print += \_itoa(t\_Delay,bufD,10);

pretty\_print += ", Total Power Consumption = ";

pretty\_print += \_itoa(t\_Power,bufP,10);

pretty\_print += "\n\n";

return pretty\_print;

}

The testbench that was created for this part of the assignment was written in a separate file:

*shiftRegTB.cpp*

This file is simplistic in nature, and opens different I/O channels to which data can be read and written. Here is the code for the testbench that works in tandem with *ShiftRegister.cpp*:

// Carlos Lazo

// ECE579D

// Homework 04

#include "ShiftRegister.h"

// Create main testbench for the ShiftRegister:

int main ()

{

string inVec;

ShiftRegister UUT;

ifstream finp ("indata.tst");

ofstream fout ("outdata.tst");

char stop\_in ('0');

finp >> inVec; // Read in first line of testbench data

while (stop\_in != '.')

{

UUT.load\_input(inVec); // Load inputs into ShiftRegister

UUT.update\_reg(); // Compute operation based on mode

UUT.set\_output(); // Set DFFs based on computed values

fout << UUT.output\_reg(); // Output data to external file

// Gather next data input:

finp >> inVec;

stop\_in = inVec[0];

}

fout << "END FILESTREAM";

return 0;

}

Each line of the *indata.tst* file contains 6 different pieces of information for each and every clock cycle of the testbench: mode select, sri, sli, din, rst, and clk. The D Flip-Flop sets the current D variables when the clock is set to a value of P. The following is a list of events that the Testbench sequentially verifies:

1. Shift Register is reset from values of ‘X’ to ‘0’
2. Parallel Load of 00001111
3. Right shift with sri = 1
4. Right shift with sri = 1
5. Left shift with sli = 0
6. *Attempted* Left shift with sli = 0, but clk = N so should not occur
7. *Attempted* Parallel Load of 00001111, but rst = 1 so rst takes precedence
8. Parallel Load of 10011001

The layout of the *indata.tst* file, as described in *ShiftRegister.cpp*, is as follows:

m1, m0, sri, srl ,din [7:0], rst, clk 🡺 14 data inputs per line

The following are the contents of *indata.tst*, and map directly to the Testbench events listed above:

**indata.tst** \* Reminder : line = m1, m0, sri, srl ,din [7:0], rst, clk

0000000000001P

1100000011110P

0110000011110P

0110000011110P

1010000011110P

1010000011110N

1110000011111P

1110100110010P

...

When running the *shiftRegTestbench.cpp* file, output is generated in the following format:

Current ShiftRegister contains the following values: <current contents>

Input = ????????, Mode = MM, Clk = C, Rst = R, sri = R, sli = L

After operation, ShiftRegister now contains: NNNNNNNN

Total time delay = xx, Total Power Consumption = yy

<current contents> = list current values located in the Shift Register (pre-mode operation)

???????? = input based on input file

MM = mode, C = clock pulse, R = reset value

R = serial right shift value, L = serial left shift value

NNNNNNNN = new values after mode change has been applied for that clock

xx = Total time delay of that operation, yy = total power consumption of that operation

Comparisons will be made between current and new ShiftRegister contents to verify operation. Upon running the testbench with the input stream above, the following output is generated:

**outdata.tst**

Current ShiftRegister contains the following values: XXXXXXXX

Input = 00000000, Mode = 00, Clk = P, Rst = 1, sri = 0, sli = 0

After operation, ShiftRegister now contains: 00000000

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 00000000

Input = 00001111, Mode = 11, Clk = P, Rst = 0, sri = 0, sli = 0

After operation, ShiftRegister now contains: 00001111

Total time delay = 44, Total Power Consumption = 16

Current ShiftRegister contains the following values: 00001111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10000111

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 10000111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 11000011

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 11000011

Input = 00001111, Mode = 10, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10000110

Total time delay = 63, Total Power Consumption = 16

Current ShiftRegister contains the following values: 10000110

Input = 00001111, Mode = 10, Clk = N, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10000110

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegister contains the following values: 10000110

Input = 00001111, Mode = 11, Clk = P, Rst = 1, sri = 1, sli = 0

After operation, ShiftRegister now contains: 00000000

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 00000000

Input = 10011001, Mode = 11, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10011001

Total time delay = 44, Total Power Consumption = 16

END FILESTREAM

Noteworthy points of information from the above output:

* Worst Case Time Delay = **70**
* Worst Case Power Consumption = **16**

Based on the circuit diagram shown above, exactly 16 gates will be hit (2 unique/DFF), in order to received resolution for each Q output, so this makes perfect sense.

A quick analysis of the output verifies the testbench events listed previously:

@ **Block 1**: ShiftRegister is reset, so all values go from ‘X’ to ‘0’

@ **Block 2**: Parallel Load of input value into ShiftRegister

@ **Block 3**: Perform right shift, with MSB = sri = 1

@ **Block 4**: Perform right shift, with MSB = sri = 1

@ **Block 5**: Perform left shift, with LSB = sli = 0

@ **Block 6**: clk = N, so effectively do nothing – ShiftRegister contents stay the same

@ **Block 7**: Even though mode = Parallel Load, clr = 1 forces ShiftRegister to clear

@ **Block 8**: Parallel Load of input value into Shift Register

The worst case delays will be used in the next portion to denote those new times.

**Huffman Shift Register & Testbench** (Parts E & F)

Given the original model of my Shift Register, the Huffman model quite is simple – all relevant code for both parts of this assignment can be found in the *shiftRegHuff* project folder in the .ZIP provided. I will only main different parts of the code that changed.

Three new files were added to the project: *shiftRegHuff.h, shiftRegHuff.cpp*, and *shiftRegTBHuff.cpp*. A new class, called *CombLogic*, was created in the *shiftRegHuff.h* file, and will simply house the combinational portions of the Shift Register. Note that the entire Shift Register declaration was changed to public, to allow for easy access to all new variable types. The following are the primary differences between the previous project and this new one:

**… <*shiftRegHuff.h*>**

// Simple class to contain combinational logic for ShiftRegister

class CombLogic {

public:

void update\_reg(ShiftRegHuff&);

};

**… <*shiftRegHuff.cpp*>**

// Huffamn Model - Combinational Portion of Shift Register

void CombLogic::update\_reg(ShiftRegHuff& out)

{

char wv; int wd, wp;

// Begin analyzing mode logic -

if ( (out.m1 == '0') && (out.m0 == '0') )

{

// Mode 0 - Do nothing. Set dTemp values to current DFF values

for (int i = 0; i < 8; i++)

out.dTemp[i] = out.reg[i].get\_Q();

}

if ( (out.m1 == '0') && (out.m0 == '1') )

{

// Mode 1 - Right Shift

for (int i = 0; i < 7; i++){

out.reg[i].get\_Q().get(wv,wd,wp);

out.dTemp[i+1].put(wv,wd,wp); } // Shift all elements to the right

out.dTemp[0].put(out.sri,0,0); // Set MSB to the sli input

}

if ( (out.m1 == '1') && (out.m0 == '0') )

{

// Mode 2 - Left Shift

for (int i = 0; i < 7; i++){

out.reg[i+1].get\_Q().get(wv,wd,wp);

out.dTemp[i].put(wv,wd,wp); } // Shift all elements to the left

out.dTemp[7].put(out.sli,0,0); // Set LSB to the sli input

}

if ( (out.m1 == '1') && (out.m0 == '1') )

{

// Mode 3 - Parallel Load

for (int i = 0; i < 8; i++)

out.dTemp[i].put(out.din[i],0,1); // Set elements to load input

}

out.t\_Delay = 70; // Back annotate worst case timing delay

out.t\_Power = 16; // Back annotate worst case power consumption

}

The testbench itself is fairly simple – both the *ShiftRegister* from part C and the *ShiftRegHuff* from part E are instantiated, and are fed the same testbench data from part D. The modified testbench code is seen below, and accounts for two different Units Under Test (UUTs):

// Carlos Lazo

// ECE579D

// Homework 04

#include "ShiftRegister.h"

#include "ShiftRegHuff.h"

// Create main testbench for the ShiftRegister:

int main ()

{

string inVec1, inVec2;

ShiftRegister UUT1;

ShiftRegHuff UUT2;

ifstream finp1 ("indata1.tst");

ofstream fout1 ("outdata1.tst");

ifstream finp2 ("indata2.tst");

ofstream fout2 ("outdata2.tst");

char stop\_in1 ('0');

char stop\_in2 ('0');

finp1 >> inVec1; // Read in first line of testbench data

while (stop\_in1 != '.')

{

UUT1.load\_input(inVec1); // Load inputs into ShiftRegister

UUT1.update\_reg(); // Compute operation based on mode

UUT1.set\_output(); // Set DFFs based on computed values

fout1 << UUT1.output\_reg(); // Output data to external file

// Gather next data input:

finp1 >> inVec1;

stop\_in1 = inVec1[0];

}

fout1 << "END FILESTREAM";

/// Now for the Huffman Model (with the worst case delays):

finp2 >> inVec2; // Read in first line of testbench data

CombLogic CL; // Define combinational logic class

while (stop\_in2 != '.')

{

UUT2.load\_input(inVec2); // Load inputs into ShiftRegister

CL.update\_reg(UUT2);

UUT2.set\_output(); // Set DFFs based on computed values

fout2 << UUT2.output\_reg(); // Output data to external file

// Gather next data input:

finp2 >> inVec2;

stop\_in2 = inVec2[0];

}

return 0;

}

In applying the testbench data to the original *ShiftRegister* UUT1, the following output it seen:

**outdata1.tst**

Current ShiftRegister contains the following values: XXXXXXXX

Input = 00000000, Mode = 00, Clk = P, Rst = 1, sri = 0, sli = 0

After operation, ShiftRegister now contains: 00000000

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 00000000

Input = 00001111, Mode = 11, Clk = P, Rst = 0, sri = 0, sli = 0

After operation, ShiftRegister now contains: 00001111

Total time delay = 44, Total Power Consumption = 16

Current ShiftRegister contains the following values: 00001111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10000111

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 10000111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 11000011

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 11000011

Input = 00001111, Mode = 10, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10000110

Total time delay = 63, Total Power Consumption = 16

Current ShiftRegister contains the following values: 10000110

Input = 00001111, Mode = 10, Clk = N, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10000110

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegister contains the following values: 10000110

Input = 00001111, Mode = 11, Clk = P, Rst = 1, sri = 1, sli = 0

After operation, ShiftRegister now contains: 00000000

Total time delay = 56, Total Power Consumption = 16

Current ShiftRegister contains the following values: 00000000

Input = 10011001, Mode = 11, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegister now contains: 10011001

Total time delay = 44, Total Power Consumption = 16

END FILESTREAM

In applying the testbench data to the original *ShiftRegHuff* UUT2, the following output it seen:

**outdata2.tst**

Current ShiftRegHuff contains the following values: XXXXXXXX

Input = 00000000, Mode = 00, Clk = P, Rst = 1, sri = 0, sli = 0

After operation, ShiftRegHuff now contains: 00000000

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 00000000

Input = 00001111, Mode = 11, Clk = P, Rst = 0, sri = 0, sli = 0

After operation, ShiftRegHuff now contains: 00001111

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 00001111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10000111

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 10000111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 11000011

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 11000011

Input = 00001111, Mode = 10, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10000110

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 10000110

Input = 00001111, Mode = 10, Clk = N, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10000110

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 10000110

Input = 00001111, Mode = 11, Clk = P, Rst = 1, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 00000000

Total time delay = 70, Total Power Consumption = 16

Current ShiftRegHuff contains the following values: 00000000

Input = 10011001, Mode = 11, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10011001

Total time delay = 70, Total Power Consumption = 16

As expected, the operation of both Shift Register implementations is exactly the same (the output matches), but the *ShiftRegHuff* has the worst case time delays and power consumption as asked. This is exactly what is expected.

**Templated Shift Register & Testbench** (Part G)

In this final part of the HW, it is asked to turn the Combinational portion of the Shift Register in Part F into a template class whose values are specialized by the delays. The code changes, which can be found in the *srHuffTemplate* project folder in the .ZIP can be found below:

**… <*shiftRegHuff.h*>**

template <class T>

class CombLogic {

T def\_d, def\_p;

public:

CombLogic (T d, T p) {def\_d = d; def\_p = p;}

void update\_reg(ShiftRegHuff&);

};

**… <*shiftRegHuff.cpp*>**

// Huffamn Model - Combinational Portion of Shift Register

void CombLogic<int>::update\_reg(ShiftRegHuff& out)

{

char wv; int wd, wp;

// Begin analyzing mode logic -

if ( (out.m1 == '0') && (out.m0 == '0') )

{

// Mode 0 - Do nothing. Set dTemp values to current DFF values

for (int i = 0; i < 8; i++)

out.dTemp[i] = out.reg[i].get\_Q();

}

if ( (out.m1 == '0') && (out.m0 == '1') )

{

// Mode 1 - Right Shift

for (int i = 0; i < 7; i++){

out.reg[i].get\_Q().get(wv,wd,wp);

out.dTemp[i+1].put(wv,wd,wp); } // Shift all elements to the right

out.dTemp[0].put(out.sri,0,0); // Set MSB to the sli input

}

if ( (out.m1 == '1') && (out.m0 == '0') )

{

// Mode 2 - Left Shift

for (int i = 0; i < 7; i++){

out.reg[i+1].get\_Q().get(wv,wd,wp);

out.dTemp[i].put(wv,wd,wp); } // Shift all elements to the left

out.dTemp[7].put(out.sli,0,0); // Set LSB to the sli input

}

if ( (out.m1 == '1') && (out.m0 == '1') )

{

// Mode 3 - Parallel Load

for (int i = 0; i < 8; i++)

out.dTemp[i].put(out.din[i],0,1); // Set elements to load input

}

// Delays are now defined by the templated class variables

out.t\_Delay = this->def\_d;

out.t\_Power = this->def\_p;

}

**… <*srTBHuffTemplate.cpp*>**

fout1 << "END FILESTREAM";

/// Now for the Huffman Model (with templated delays):

finp2 >> inVec2; // Read in first line of testbench data

CombLogic<int> CL (13,37); // Define TEMPLATED combinational logic class

while (stop\_in2 != '.')

{

UUT2.load\_input(inVec2); // Load inputs into ShiftRegister

CL.update\_reg(UUT2);

UUT2.set\_output(); // Set DFFs based on computed values

fout2 << UUT2.output\_reg(); // Output data to external file

// Gather next data input:

finp2 >> inVec2;

stop\_in2 = inVec2[0];

}

return 0;

}

In this new definition, it is seen that the CombLogic class is being template instantiated with a delay value of 13 and a power consumption value of 37. When instantiating this UUT with the testbench in Part F, we must verify that the new data matches the templated integers provided:

In applying the testbench data to the *ShiftRegHuff* UUT2 with templates, the following output it seen:

**outdata2.tst**

Current ShiftRegHuff contains the following values: XXXXXXXX

Input = 00000000, Mode = 00, Clk = P, Rst = 1, sri = 0, sli = 0

After operation, ShiftRegHuff now contains: 00000000

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 00000000

Input = 00001111, Mode = 11, Clk = P, Rst = 0, sri = 0, sli = 0

After operation, ShiftRegHuff now contains: 00001111

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 00001111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10000111

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 10000111

Input = 00001111, Mode = 01, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 11000011

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 11000011

Input = 00001111, Mode = 10, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10000110

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 10000110

Input = 00001111, Mode = 10, Clk = N, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10000110

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 10000110

Input = 00001111, Mode = 11, Clk = P, Rst = 1, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 00000000

Total time delay = 13, Total Power Consumption = 37

Current ShiftRegHuff contains the following values: 00000000

Input = 10011001, Mode = 11, Clk = P, Rst = 0, sri = 1, sli = 0

After operation, ShiftRegHuff now contains: 10011001

Total time delay = 13, Total Power Consumption = 37

In looking at the data above, it is clear that the template Combinational Logic portion of the Shift Register did, in fact, take the integer values it was assigned through the use of C++ templates.

This concludes the analysis of Homework 04.